DISCUSSING ASPECTS OF AOP

Tzilla Elrad, Moderator
Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher, Panelists

How would you define AOP?

Gregor Kiczales: Aspect-oriented programming is a new evolution in the line of technology for separation of concerns—technology that allows design and code to be structured to reflect the way developers want to think about the system. AOP builds on existing technologies and provides additional mechanisms that make it possible to affect the implementation of systems in a crosscutting way. In AOP, a single aspect can contribute to the implementation of a number of procedures, modules, or objects. The contribution can be homogeneous, for example by providing a logging behavior that all the procedures in a certain interface should follow; or it can be heterogeneous, for example by implementing the two sides of a protocol between two different classes.

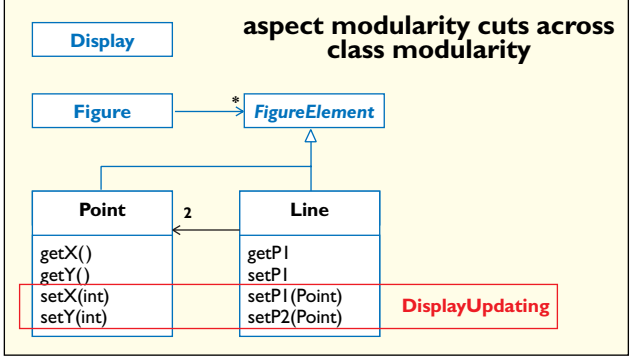
As with all other separation of concerns technology, the

goal of AOP is to make designs and code more modular, meaning the concerns are localized rather than scattered and have well-defined interfaces with the rest of the system. This provides us with the usual benefits of modularity, including making it possible to reason about different concerns in relative isolation, making them (un)pluggable, amenable to separate development, and so forth.

Please say more about the nature of crosscutting concerns and aspects.

Karl Lieberherr: Two concerns crosscut if the methods related to those concerns intersect. AOP deals with crosscutting concerns and descriptions, designs, and implementations for those concerns. The artifacts used to describe, design, and implement a given concern are called methods. We say a method is related to a concern if the method contributes to the description, design, or implementation of the concern.

Aspects crosscut classes in a simple figure editor.



GK: One good way to understand crosscutting concerns and aspects is with an illustration. Consider the UML for a simple figure editor, as depicted in the figure here, in which there are two concrete classes of figure element, points, and lines. These classes manifest good modularity, in that the source code in each class is closely related (cohesion) and each class has a clear and well-defined interface. But consider the concern that the screen manager should be notified whenever a figure element moves. This requires every method that moves a figure element to do the notification.

The red box in the figure is drawn around every method that must implement this concern, just as the Point and Line boxes are drawn around every method that implements those concerns. Notice that the box for DisplayUpdating fits neither inside of nor around the other boxes in the figure—instead it cuts across the other boxes. This is what we call a crosscutting concern. Using just OO programming, the implementation of crosscutting concerns tends to be scattered out across the system, just as it would be here. Using the mechanisms of AOP, we can modularize the implementation of the DisplayUpdating behavior into a single aspect. Because we can implement this behavior in a single modular unit, it makes it easier for us to think about it as a single design unit. In this way, having the programming language mechanisms of aspects lets us think in terms of aspects at the design level as well.

Mehmet Aksit: It is important to understand that crosscutting is relative to a particular decomposition. Crosscutting concerns of a design cannot be neatly separated from each other. A basic design rule is to represent significant concerns as first-class abstractions in the language. This allows them to be composed and extended. In the figure editor example, there are two important design concerns: representing the graphical elements and tracking the movement of graphical elements. In the figure, classes are used to model the first concern. This allows them to be extended using aggregation and inheritance. Also, every graphical class encapsulates its internal data structure. The second concern requires tracking movements to also be represented as a separate class. However, the first choice makes this difficult because the movement functionality is part of the behavior of graphical classes. We could have designed the system around the tracking functionality; in that case, the graphical functionality would crosscut the tracking classes

Harold Ossher: One of the hard things about crosscutting concerns is understanding just what cuts across what. To clarify this, I think the dominant decomposition notion is helpful. Software written in standard languages is written as linear text. This means that, just as a book is divided in only one way into chapters and paragraphs, so software is decomposed in only one way into modules (such as classes). This the dominant decomposition. The modules making up the dominant decomposition encapsulate certain concerns effectively (representation and implementation details of objects of some kind are encapsulated by classes). As noted by others previously, other concerns cannot be encapsulated within the dominant modules, and end up being scattered across many modules and tangled with one another. These are crosscutting concerns.

How do AOP languages make it possible to modularize crosscutting concerns?

KL: AOP languages use five main elements to modularize crosscutting concerns: a join point model describing the "hooks" where enhancements may be added; a means of identifying join points; a means of specifying behavior at join points; encapsulated units combining join point specifications and behavior enhancements; and a method of attachment of units to a program.

HO: I'll describe the approach we use in Hyper/J, which supports identification and modularization of arbitrary concerns, including crosscutting concerns. Hyper/J provides a new kind of module, called hyperslices (previously called subjects in subject-oriented programming). Each hyperslice is a fragment of a class hierarchy—a hierarchy of classes—but with each class containing only the subset of methods and variables specifically pertaining to the concern being modularized.

Referring back to the figure, we would modularize the concerns in two hyperslices: a Display hyperslice containing the classes shown, but with just their basic display behavior, and a DisplayUpdating hyperslice containing fragments of the Point and Line classes. A simple (but not the best) way to write this DisplayUpdating hyperslice is to implement each method shown in the figure to do just the tracking needed. A hyperslice can thus be thought of as a view, or a special-purpose domain model, appropriate for the concern at hand. Hyperslices can be written from scratch as separate Java packages. They can also be extracted from existing code, within limits, thereby modularizing concerns that were not modularized in the original code, a process we call on-demand remodularization. Systems are built from hyperslices by composition, discussed later, which results in the synthesis of complete class hierarchies from the fragments, including the combination of behavior.

How do AOP languages use join points?

GK: In AspectJ the join points are well-defined points in the execution of the program including method calls, field accesses, and object construction. AspectJ makes it possible to give names to sets of join points and to associate additional implementation that should run before, after, or around those events. AspectC works in a similar fashion.

KL: We think of join points as nodes or edges in some graph. The graph can be a dynamic call graph (a UML interaction diagram), a class graph (a UML class diagram), an object graph (a UML object diagram). For example, in a simple use of the DJ library, the join point model consists of object graphs. And we add behavior to those points using visitor objects that add code to both nodes and edges. An appealing way to define join points is to use succinct specifications: Instead of enumerating the join points, we use a short description that can be filled in with information from elsewhere to create the list of join points. In adaptive programming, join points are defined by succinct specifications called traversal strategies that, if complemented by a class diagram, define the detailed list of join points. Using traversal strategies leads to loose coupling between the structural and the numerous behavioral concerns.

HO: In Hyper/J, composition involves finding corresponding join points in the hyperslices being composed and combining the hyperslices at those join points. Join points include classes, interfaces, methods, and member variables. Correspondences and details of composition are specified by composition relationships supplied by the developer. These are often simple and general, such as "mergeByName" which means "join points with the same names (and signatures) in different hyperslices correspond, and combination is done by merging." More specific relationships are also available, to deal with such matters as correspondence despite name differences, and order of execution and return value computation for merged methods. Examples are given in our article in this section, which also shows an additional use of join points: the ability to pull software apart at join points, extracting separate hyperslices from a tangled class hierarchy.

MA: The Composition Filters model uses the join points to attach concerns to messages that are received or sent by objects. The important objectives of this model are to minimize the interference among concerns that are attached to the same join point, to express concerns in a language-independent way, and to provide explicit operators for composing concerns.

Does AOP replace OOP?

GK: Absolutely not! When we moved to OOP we did not throw out procedures and all we had learned about using them. Similarly, the goal for AOP is to build on OOP, by supporting separation of those concerns that OOP handles poorly. When programming in AOP we use procedures, objects, and aspects, each when most appropriate. And we should say right now that AOP will not be the last word on the problem of separation of concerns either. In the future we can expect to see important new developments in this area.

HO: Following on Gregor's remark, Hyper/J is based on an approach we call multidimensional separation of concerns, which takes some of the ideas even further, and which we are beginning to apply to concerns across the life cycle. It supplements OO (or, potentially, other paradigms), but definitely does not replace it. In addition to supporting modularization of crosscutting concerns, MDSOC allows multiple, different decompositions of a system to coexist, and new ones to be introduced as needed (on-demand remodularization). For example, one developer might work with a class, encapsulating a data concern (all representation details of an abstract data type). Another might work with a hyperslice encapsulating a "feature concern:" a collection of members of different classes that, together, implement that feature. Yet another might work with a hyperslice encapsulating a business rule concern: a collection of members of different

classes that, together, check and enforce some business rule. Although they overlap, since they carve up the system in different ways, all these hyperslices can coexist as modules. This finally breaks away from programs-as-linear-text; we call it overthrowing the tyranny of the dominant decomposition.

MA: Using Composition Filters extends OOP by encoding concerns as manipulators of messages that are received or sent by objects. Composition Filters builds on the native message-passing mechanism among objects and leaves the expression of noncrosscutting concerns to the implementation of objects. It integrates AOP and OOP in a complementary way.

Key concepts compared.

Technology	Key Concepts	Constructs
Structured programming	Explicit control constructs	Do, while and other loops, blocks, and so forth
Modular programming	Information hiding	Modules with well-defined enforced interfaces
Data abstraction	Hide the representation of data	Types
Object-oriented programming	Objects, with classification and specialization	Classes, objects, polymorphism

What do you think is (are) the key issue(s)?

KL: A key issue is the reusability of aspects. To make aspects more reusable, we introduced the concept of aspectual collaborations, a derivative of AP&PC presented at OOPSLA 1998. An aspectual collaboration describes an aspect using a class graph. When the collaboration is used, the class graph is mapped into a larger class graph using an adapter. Aspectual collaborations and adapters lead to better separation of crosscutting issues expressed in adapters and reusable behavior expressed in aspectual collaborations. It is not good enough to modularize crosscutting concerns because the modularization might scatter another concern leading to a program that is still hard to maintain. We need to modularize crosscutting concerns such that they are loosely coupled to other parts of the program.

GK: Reusability of aspects is certainly one key issue. AspectJ includes reuse mechanisms similar to those found in OOP. We know these are useful for making small reusable aspects. Now we need to develop expertise in making larger libraries of reusable aspects. A related issue is how to work with large numbers of aspects. How do they compose? How do we think about such designs? What notations should we use for describing them? These and other related questions are ones we expect the user and research communities to explore over the next few years.

MA: According to our viewpoint, the key issues of AOP can be summarized using the following six "C"s—Crosscutting, Canonicality, Composability, Computability, Closure property, and Certifiability:

- The crosscutting property has been explained;
- The canonical property is necessary for the stability of the implementation of concerns;
- The composability property is necessary for providing quality factors such as adaptability, reusability, and extensibility;
- The computability property is necessary for creating executable software systems;
- The closure property is necessary for maintaining the quality factors of the design at the implementation level; and
- The certifiability property is necessary for evaluating and controlling the quality of design and implementation models.

HO: Along with Mehmet's six "C"s, we have four "S"s for successful separation:

- Simultaneous: coexistence of different decompositions, as described earlier, are important.
- Self-contained: each module should declare what it depends on, so that it can be understood in isolation. In Hyper/J, we do this using an approach called declarative completeness.
- Symmetric: there should be no distinction in form between the modules encapsulating different kinds of concerns, so that they can be composed together most flexibly. This is related to Mehmet's composability and closure properties. For example, we want aspects to be able to extend other aspects as well as classes.
- Spontaneous: it should be possible to identify and encapsulate new concerns, and even new kinds of concerns, as they arise during the software life cycle.

How can OO software engineers benefit from your approach?

KL: We have implemented the DJ library that allows Java programmers to write an important class of functional aspects directly in Java. In addition, the DJ library is a good tool to define how to reuse an aspect (class graph mapping). The DJ library is a very user-friendly implementation of adaptive programming and the loose coupling between structure and behavior that it offers has an impact on the maintenance cost of the software.

HO: There is a spectrum of scenarios in which our

approach can provide benefit, some of them also supported by other approaches. These include:

- Writing packages in standard Java, to encapsulate functional or nonfunctional crosscutting concerns. This includes modular insertion of code, such as instrumentation, monitoring or debugging code, into existing classes.
- Extraction and modularization of the code pertaining to concerns not separated out when the code was written. This can be thought of as post-hoc, on-demand separation.
- Composing reusable concerns with custom software.
- Multiteam development, where each team evolves its own domain model as its work proceeds, and these models need to be reconciled and integrated.
- Integration of separately written applications and domain models, possibly using glue code. Details of the integration are specified in the composition relationships.

Many research challenges remain in realizing some of these scenarios.

MA: Based on our experiences in a number of pilot projects, we presented a list of obstacles that software engineers may experience if they adopt OOP languages. The so-called "lack of support for multiple views" and "synchronization and real-time composition anomalies" are some examples of the obstacles that we have listed. We also categorized the obstacles with respect to certain application domains. Software engineers may first look at these publications and determine if there are similarities between their software design problems and our pilot studies. Many of the identified obstacles have been addressed by our research work on Composition Filters. If the Composition Filters model offers a solution to their design problems, software engineers have two options: (a) They may adopt the Composition Filters model as a design-level solution. In this case, filters may be implemented, for example, by using message reflection techniques or (b) They may utilize a Composition Filters compiler to translate filter specifications to an implementation language such as Java.

GK: AspectJ is a simple extension to Java. AspectJ is designed so it can be easily integrated with existing programming practice and tools. Many AspectJ users start by writing various development process aspects like testing, tracing, logging, and contract checking. They then move on to writing aspects—using AspectJ tends to reduce development time and make software easier to debug and modify after it has been written.

What are one or two key open issues that are important to address during the post-OO era?

HO: One of the most important open issues is semantic correctness of aspects and compositions. It has always been an issue with modular systems to ensure that modules are correct on their own and that they interact correctly when composed. The join point-based composition provided by aspect languages is much richer than the interface- or message-based connection provided by most other modularization mechanisms, complicating both specification and testing of compositions. Even specification and unit testing of individual aspects is an area requiring research.

KL: An important topic is how to deal with reusable aspects. Consider an adaptive method written with the DJ Java library. It works with a large family of Java programs but for which programs does it work as intended? Consider a reusable aspect in AspectJ that involves the star operator in the pointcut designators. It also works with a large family of Java programs but again, for which programs does it work as intended? The concept of correctness of a reusable aspect is an open issue. We need to find good ways to express the assumptions that need to hold for a reusable aspect to work correctly in a specific context.

MA: The six "C"s of AOP, introduced previously, are also our challenges. First, we need to improve our understanding about the crosscutting concerns. For example, what kind of crosscutting concerns are typical in application areas such as e-commerce and what are the characteristics of these concerns? Second, we need to define new methods for identifying and specifying canonical models for crosscutting concerns. Third, we need to specify the relevant composability operations on the canonical concern models. Fourth, we should be able to define a—preferably single—translator for all possible concern specifications so that new concerns can be introduced without redefining the translator. In addition, the translator should generate efficient code. Fifth, we need to improve our understanding about the static and dynamic characteristics of crosscutting concerns so that we can maintain the functional and quality characteristics of the crosscutting concepts at runtime. Finally, we need to develop formal models for determining the functional and quality characteristics of crosscutting concerns individually and together.

Doesn't the ability to compose/weave classes break the encapsulation that is such an important feature of OO?

GK: No. AOP adds a new kind of modularity to the programmer's toolkit. Crosscutting concerns are an inherent part of most complex systems. Today programmers are being forced to write tangled code when-

ever they have to implement a crosscutting concern. With AOP, the code for crosscutting concerns is localized, the structure of the crosscutting is explicit, and the code can often be a good deal shorter as well.

HO: It is true that aspects can introduce new behavior at join points, such as private methods, that normally could not be modified except by changing the source code. However, if these aspects are considered to be part of the class(es) they extend, just written separately, then they do not break the class encapsulation. In fact, the encapsulation can be tighter (depending on the details of the aspect language): the code is encapsulated by both the aspect and the class, and need not be made visible even to other aspects of the same class. Additional visibility or security mechanisms might be appropriate in some environments to control which developers are allowed to write aspects that extend specific classes.

MA: Crosscutting concerns in the Composition Filters model do not depend upon the implementation details, such as attributes and private methods, of other concerns. One of the important characteristics of the Composition Filters model is that concerns are attached to messages. Filters are modular extensions to objects; even the implementation language is encapsulated. This makes porting filters to different languages easy.

Doesn't dealing with all these concerns make the program more difficult to understand?

HO: The concerns are there in the program anyway, all tangled, and they are indeed a major source of complexity. But, unless the system is badly written, they are inherent. What these approaches do is allow the implicit, tangled concerns to be separated out and made explicit, so developers can see what they're dealing with. Just as with OO, overzealous separation can lead to so many modules that their relationships become obscure. It is still important to follow good design principles, and separate just those concerns that need to be separated. That is often hard to predict, which is why we emphasize the spontaneity property described previously.

MA: The crosscutting concerns of a design represent crucial abstractions. It is therefore important to represent these concerns explicitly, encapsulate their implementation details, and provide operations to extend them. AOP languages make crosscutting concerns more understandable and manageable.

Much of software evolution is of unanticipated nature. How does AOP help?

KL: In our work on aspectual collaborations, we use an approach similar to Hyper/J. Our aspectual collaborations are declaratively complete and talk only about "abstract" join points. Each aspectual collaboration is formulated in terms of a high-level class graph and this supports unanticipated reuse that is formulated inside the adapters. Adaptive programming is a useful tool to deal with unanticipated class diagram evolution.

HO: Graceful evolution (not involving invasive changes to existing code) requires suitable hooks. Many design patterns are intended to provide such hooks for cases of expected evolution, but it is impossible to provide explicitly all possible hooks for unanticipated evolution. Join points can come to the rescue. They provide a rich set of hooks that can be used as needed, without incurring overhead when not used. A wide class of extensions can thus be implemented as aspects, which can be attached at suitable join points noninvasively. Since unanticipated evolution was one of our primary motivations, we have taken support for it even further in Hyper/J. Sometimes the structure of the class hierarchy turns out to be unsuitable for some evolutionary task. Sometimes concerns or kinds of concerns emerge during evolution that were not separated when the code was originally written. It is necessary to allow hyperslices to have different class hierarchies (views, domain models), and to be able to extract hyperslices from existing code. Our article in this section discusses these issues and gives examples.

MA: In the Composition Filters model, aspects can be attached to or removed from objects even at runtime. Since the observable behavior of an object is largely defined by its interaction semantics, filters can extend objects in many possible ways. **C**

Tzilla Elrad (elrad@iit.edu) is a research professor leading the Concurrent Programming Research Group in the Department of Computer Science at the Illinois Institute of Technology in Chicago.
Mehmet Aksits (aksit@cs.utwente.nl) is Professor of Software Engineering and Chair of the Department of Computer Science at the University of Twente, The Netherlands.
Gregor Kiczales (gregor@cs.ubc.ca) is Professor and NSERC, Xerox, Sierra Systems Chair of Software Design at the University of British Columbia, and Principal Scientist at Xerox PARC.
Karl Lieberherr (lieber@ccs.neu.edu) is Professor of Computer Science at Northeastern University in Boston, MA.
Harold Ossher (ossher@watson.ibm.com) is a Research Staff Member at the IBM T.J. Watson Research Center in Yorktown Heights, NY.